1. What does static variable mean?

Ans: Static variables are the variables which retain their values between the function calls. They are initialized only once their scope is within the function in which they are defined.

2. What is a pointer?

Ans: Pointers are variables which stores the address of another variable. That variable may be a scalar (including another pointer), or an aggregate (array or structure). The pointed-to object maybe part of a larger object, such as a field of a structure or an element in an array.

3. What is a structure?

Ans: Structure constitutes a super data type which represents several different data types in a single unit. A structure can be initialized if it is static or global.

4. What is a union?

Ans: Union is a collection of heterogeneous data type but it uses efficient memory utilization technique by allocating enough memory to hold the largest member. Here a single area of memory contains values of different types at different time. A union can never be initialized.

5. What are the differences between structures and union?

Ans: A structure variable contains each of the named members, and its size is large enough to hold all the members. Structure elements are of same size. A union contains one of the named members at a given time and is large enough to hold the largest member. Union element can be of different sizes.

6. What are the differences between structures and arrays?

Ans:

| **Basis for Comparison** | **Array** | **Structure** |
| --- | --- | --- |
| Basic | An array is a collection of variables of same data type. | A structure is a collection of variables of different data type. |
| Syntax | type array\_name[size]; | struct sruct\_name{ type element1; type element1; . . } variable1, variable2, . .; |
| Memory | Array elements are stored in contiguous memory location. | Structure elements may not be stored in a contiguous memory location. |
| Access | Array elements are accessed by their index number. | Structure elements are accessed by their names. |
| Operator | Array declaration and element accessing operator is "[ ]" (square bracket). | Structure element accessing operator is "." (Dot operator). |
| Pointer | Array name points to the first element in that array so, array name is a pointer. | Structure name does not point to the first element in that structure so, structure name is not a pointer. |
| Objects | Objects (instances) of an array can not be created. | Structure objects (instance or structure variable) can be created. |
| Size | Every element in array is of same size. | Every element in a structure is of different data type. |
|  |  |  |
| Keyword | There is no keyword to declare array. | "struct" is a keyword used to declare the structure. |
| User-defined | Arrays are not user-defined they are directly declared. | Structure is a user-defined datatype. |
| Accessing | Accessing array element requires less time. | Accessing a structure elements require comparatively more time. |

7. In header files whether functions are declared or defined?

Ans: Functions are declared within header file. That is function prototypes exist in a header file, not function bodies. They are defined in library (lib).

8. What are the differences between malloc () and calloc ()?

Ans:

|  |  |
| --- | --- |
| malloc() takes one argument that is, *number of bytes*. | calloc() take two arguments those are: *number of blocks* and *size of each block*. |
| syntax of malloc():  void \*malloc(size\_t n);  Allocates n bytes of memory. If the allocation succeeds, a void pointer to the allocated memory is returned. Otherwise NULL is returned. | syntax of calloc():  void \*calloc(size\_t n, size\_t size);  Allocates a contiguous block of memory large enough to hold n elements of size bytes each. The allocated region is initialized to zero. |
| malloc is faster than calloc. | calloc takes little longer than malloc because of the extra step of initializing the allocated memory by zero. However, in practice the difference in speed is very tiny and not recognizable. |
| void \*malloc(size\_t n) returns a pointer to n bytes of uninitialized storage, or NULL if the request cannot be satisfied. If the space assigned by malloc() is overrun, the results are undefined. | void \*calloc(size\_t n, size\_t size) returns a pointer to enough free space for an array of n objects of the specified size, or NULL if the request cannot be satisfied. The storage is initialized to zero. |

**9.** **What is the difference between call by value and call by reference in C language?**

In *call by value*, a copy of actual arguments is passed to formal arguments of the called function and any change made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.

In *call by reference*, the location (address) of actual arguments is passed to formal arguments of the called function. This means by accessing the addresses of actual arguments we can alter them within from the called function.

In call by value, actual arguments will remain safe, they cannot be modified accidentally.

In *call by reference*, alteration to actual arguments is possible within from called function; therefore the code must handle arguments carefully else you get unexpected results.

**10. What are macros? What are its advantages and disadvantages?**

Ans: Macros are abbreviations for lengthy and frequently used statements. When a macro is called the entire code is substituted by a single line though the macro definition is of several lines. The advantage of macro is that it reduces the time taken for control transfer as in case of function. The disadvantage of it is here the entire code is substituted so the program becomes lengthy if a macro is called several times.

**11. What is static identifier?**

Ans: A file-scope variable that is declared static is visible only to functions within that file. A function-scope or block-scope variable that is declared as static is visible only within that scope. Furthermore, static variables only have a single instance. In the case of function-or block-scope variables, this means that the variable is not “automatic” and thus retains its value across function invocations.

**12. Where is the auto variables stored?**

Ans: Auto variables can be stored anywhere, so long as recursion works. Practically, they‘re stored on the stack. It is not necessary that always a stack exist. You could theoretically allocate function invocation records from the heap.

**13. What are enumerations?**

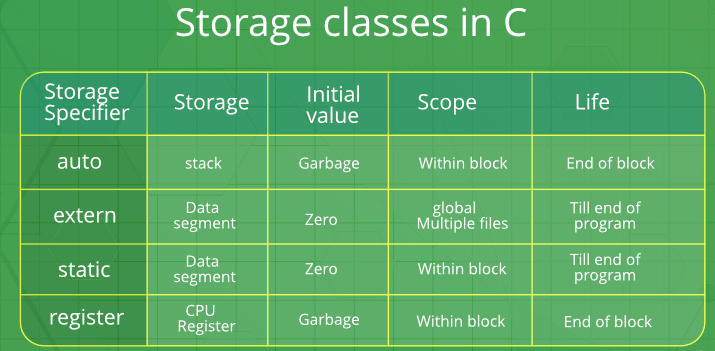
Ans: They are a list of named integer-valued constants. Example:enum color { black , orange=4, yellow, green, blue, violet };This declaration defines the symbols ”black”, “orange”, “yellow”, etc. to have the values “1”, “4”, “5”, …etc. The difference between an enumeration and a macro is that the enum actually declares a type, and therefore can be type checked.

**14. Describe about storage allocation and scope of global, extern, static, local and register variables?**

**Storage Classes in C**

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:

****

**auto:** This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

**extern:** Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

**static:** This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

**register:** This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

**15. What are register variables?**

**What are the advantages of using register variables?**

Ans: If a variable is declared with a register storage class, it is known as register variable. The register variable is stored in the cpu register instead of main memory. Frequently used variables are declared as register variable as it‘s access time is faster.

**16. What is the use of typedef?**

Ans: The typedef help in easier modification when the programs are ported to another machine. A descriptive new name given to the existing data type may be easier to understand the code.

**17. What is static memory allocation?**

Ans: Compiler allocates memory space for a declared variable. By using the address of operator, the reserved address is obtained and this address is assigned to a pointer variable. This way of assigning pointer value to a pointer variable at compilation time is known as static memory allocation.

**18. What is dynamic memory allocation?**

Ans: A dynamic memory allocation uses functions such as malloc() or calloc() to get memory dynamically. If these functions are used to get memory dynamically and the values returned by these function are assigned to pointer variables, such a way of allocating memory at run time is known as dynamic memory allocation.

**19. In C, why is the void pointer useful? When would you use it?**

Ans: The void pointer is useful because it is a generic pointer that any pointer can be cast into and back again without loss of information.

**20. What is a NULL Pointer? Whether it is same as an uninitialized pointer?**

Ans: Null pointer is a pointer which points to nothing but uninitialized pointer may point to anywhere.

**21. Are pointers integer?**

Ans: No, pointers are not integers. A pointer is an address. It is a positive number.

**22. What does the error ‘Null Pointer Assignment’ means and what causes this error?**

Ans: As null pointer points to nothing so accessing a uninitialized pointer or invalid location may cause an error.

**23. What is generic pointer in C?**

Ans: In C void\* acts as a generic pointer. When other pointer types are assigned to generic pointer, conversions are applied automatically (implicit conversion).

**24. Are the expressions arr and &arr same for an array of integers?**

Ans: Yes for array of integers they are same.

**25. What is static memory allocation?**

Ans: Compiler allocates memory space for a declared variable. By using the address of operator, the reserved address is obtained and this address is assigned to a pointer variable. This way of assigning pointer value to a pointer variable at compilation time is known as static memory allocation.

**26. What is pointer to a pointer?**

Ans: If a pointer variable points another pointer value. Such a situation is known as a pointer to a pointer.

Example:

int \*p1,\*\*p2,v=10;

P1=&v;

p2=&p1;

Here p2 is a pointer to a pointer.

**27. What is an array of pointers?**

Ans: if the elements of an array are addresses, such an array is called an array of pointers.

**28. Difference between linker and linkage?**

Ans: Linker converts an object code into an executable code by linking together the necessary built in functions. The form and place of declaration where the variable is declared in a program determine the linkage of variable.

**29. Is it possible to have negative index in an array?**

Ans: Yes it is possible to index with negative value provided there are data stored in this location. Even if it is illegal to refer to the elements that are out of array bounds, the compiler will not produce error because C has no check on the bounds of an array.

**30. Why is it necessary to give the size of an array in an array declaration?**

Ans: When an array is declared, the compiler allocates a base address and reserves enough space in memory for all the elements of the array. The size is required to allocate the required space and hence size must be mentioned.

**31. What is a function?**

Ans: A large program is subdivided into a number of smaller programs or subprograms. Each subprogram specifies one or more actions to be performed for the larger program. Such sub programs are called functions.

**32. What is an argument?**

Ans: An argument is an entity used to pass data from the calling to a called function.

**33. What are built in functions?**

Ans: The functions that are predefined and supplied along with the compiler are known as built in functions. They are also known as library functions.

**34. Difference between formal argument and actual argument?**

Ans: Formal arguments are the arguments available in the function definition. They are preceded by their own data type. Actual arguments are available in the function call. These arguments are given as constants or variables or expressions to pass the values to the function.

**35. Is it possible to have more than one main() function in a C program ?**

Ans: The function main() can appear only once. The program execution starts from main.

**36. What is FILE?**

Ans: FILE is a predefined data type. It is defined in stdio.h file.

**37. What is a file pointer?**

Ans: The pointer to a FILE data type is called as a stream pointer or a file pointer. A file pointer points to the block of information of the stream that had just been opened.

**38. How is fopen() used ?**

Ans: The function fopen() returns a file pointer. Hence a file pointer is declared and it is assigned as FILE \*fp;fp= fopen(filename,mode);filename is a string representing the name of the file and the mode represents:―

“r" for read operation

“w" for write operation

“a” for append operation

“r+”, “w+”, “a+” for update operation.

**39. How is a file closed ?**

Ans: A file is closed using fclose() function

Eg. fclose(fp);

Where fp is a file pointer.

**40. Are the expressions \*ptr ++ and ++ \*ptr same?**

Ans: No,\*ptr ++ increments pointer and not the value pointed by it. Whereas ++ \*ptr increments the value being pointed to by ptr.